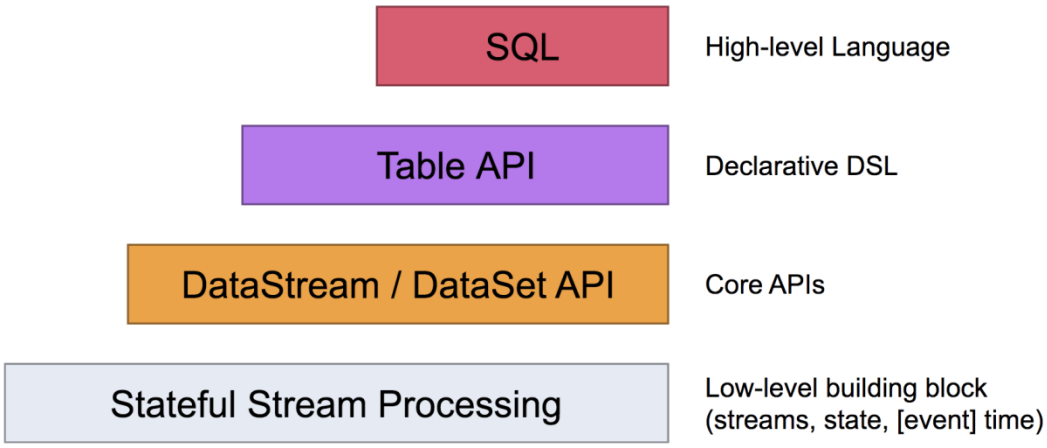
Flink SQL 编程

（作者：尚硅谷大数据研发部）

版本：V1.12.0

如果用户需要同时流计算、批处理的场景下，用户需要维护两套业务代码，开发人员也要维护两套技术栈，非常不方便。

Flink 社区很早就设想过将批数据看作一个有界流数据，将批处理看作流计算的一个特例，从而实现流批统一，Flink 社区的开发人员在多轮讨论后，基本敲定了 Flink 未来的技术架构



Apache Flink 有两种关系型 API 来做流批统一处理：Table API 和 SQL。

Table API 是用于 Scala 和 Java 语言的查询 API，它可以用一种非常直观的方式来组合使用选取、过滤、join 等关系型算子。

Flink SQL 是基于 Apache Calcite 来实现的标准 SQL。这两种 API 中的查询对于批（DataSet）和流（DataStream）的输入有相同的语义，也会产生同样的计算结果。

Table API 和 SQL 两种 API 是紧密集成的，以及 DataStream 和 DataSet API。你可以在这些 API 之间，以及一些基于这些 API 的库之间轻松的切换。比如，你可以先用 CEP 从 DataStream 中做模式匹配，然后用 Table API 来分析匹配的结果；或者你可以用 SQL 来扫描、过滤、聚合一个批式的表，然后再跑一个 Gelly 图算法 来处理已经预处理好的数据。

更多 Java –大数据 –前端 –python 人工智能资料下载，可百度访问：尚硅谷官网

注意：Table API 和 SQL 现在还处于活跃开发阶段，还没有完全实现所有的特性。不是所有的 [Table API，SQL] 和 [流，批] 的组合都是支持的。

第1章 核心概念

Flink 的 Table API 和 SQL 是流批统一的 API。这意味着 TableAPI & SQL 在无论有限的批式输入还是无限的流式输入下，都具有相同的语义。因为传统的关系代数以及 SQL 最开始都是为了批式处理而设计的，关系型查询在流式场景下不如在批式场景下容易理解。

1.1 动态表和连续查询

*动态表(Dynamic Tables)*是 Flink 的支持流数据的 Table API 和 SQL 的核心概念。

与表示批处理数据的静态表不同，动态表是**随时间变化**的。可以像查询静态批处理表一样查询它们。查询动态表将生成一个*连续查询(Continuous Query)*。一个连续查询永远不会终止，结果会生成一个动态表。查询不断更新其(动态)结果表，以反映其(动态)输入表上的更改。

需要注意的是，连续查询的结果在语义上总是等价于以批处理模式在输入表快照上执行的相同查询的结果。

1. 将流转换为动态表。
2. 在动态表上计算一个连续查询，生成一个新的动态表。
3. 生成的动态表被转换回流。

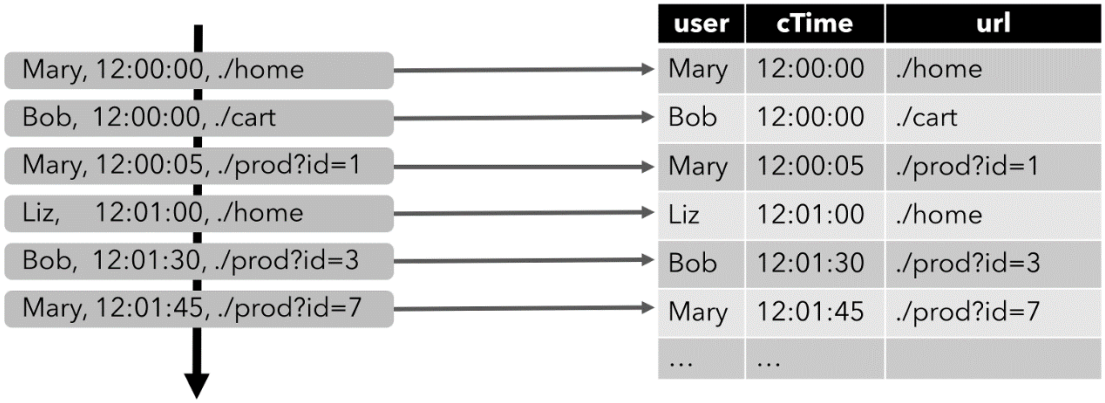
1.2 在流上定义表(动态表)

为了使用关系查询处理流，必须将其转换成 Table。从概念上讲，流的每条记录都被解释为对**结果表的** INSERT 操作。

假设有如下格式的数据:

```
[
  user:  VARCHAR,   // 用户名
  cTime: TIMESTAMP, // 访问 URL 的时间
  url:   VARCHAR    // 用户访问的 URL
]
```

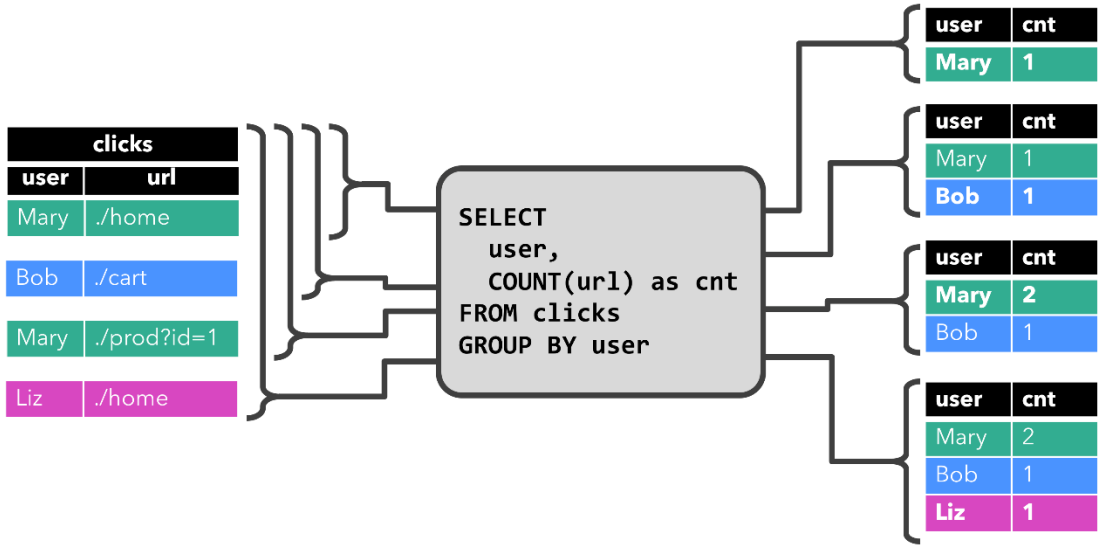
下图显示了单击事件流(左侧)如何转换为表(右侧)。当插入更多的单击流记录时，结果表的数据将不断增长。



1.2.1 连续查询

在动态表上计算一个连续查询，并生成一个新的动态表。与批处理查询不同，连续查询从不终止，并根据其输入表上的更新更新其结果表。

在任何时候，连续查询的结果在语义上与以批处理模式在输入表快照上执行的相同查询的结果相同。



说明：

1. 当查询开始，clicks 表(左侧)是空的。
2. 当第一行数据被插入到 clicks 表时，查询开始计算结果表。第一行数据 [Mary,./home] 插入后，结果表(右侧，上部)由一行 [Mary, 1] 组成。
3. 当第二行 [Bob, ./cart] 插入到 clicks 表时，查询会更新结果表并插入了

一行新数据 [Bob, 1]。

4. 第三行 [Mary, ./prod?id=1] 将产生已计算的结果行的更新，[Mary, 1] 更新成 [Mary, 2]。
5. 最后，当第四行数据加入 clicks 表时，查询将第三行 [Liz, 1] 插入到结果表中。

第2章 Flink Table API

2.1 导入需要的依赖

```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-table-planner-blink_${scala.binary.version}</artifactId>
    <version>${flink.version}</version>
</dependency>
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-streaming-scala_${scala.binary.version}</artifactId>
    <version>${flink.version}</version>
</dependency>
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-csv</artifactId>
    <version>${flink.version}</version>
</dependency>
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-json</artifactId>
    <version>${flink.version}</version>
</dependency>
```

2.2 基本使用:表与 DataStream 的混合使用


```
package com.atguigu.flink.java.chapter_11;

import com.atguigu.flink.java.chapter_5.WaterSensor;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.datastream.DataStreamSource;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.table.api.Table;
import org.apache.flink.table.api.bridge.java.StreamTableEnvironment;
import org.apache.flink.types.Row;

import static org.apache.flink.table.api.Expressions.$;

public class Flink01_TableApi_BasicUse {
    public static void main(String[] args) {
```

```
        StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(1);
        DataStreamSource<WaterSensor> waterSensorStream =
            env.fromElements(new WaterSensor("sensor_1", 1000L, 10),
                             new WaterSensor("sensor_1", 2000L, 20),
                             new WaterSensor("sensor_2", 3000L, 30),
                             new WaterSensor("sensor_1", 4000L, 40),
                             new WaterSensor("sensor_1", 5000L, 50),
                             new WaterSensor("sensor_2", 6000L, 60));

        // 1. 创建表的执行环境
        StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env);
        // 2. 创建表: 将流转换成动态表. 表的字段名从pojo的属性名自动抽取
        Table table = tableEnv.fromDataStream(waterSensorStream);
        // 3. 对动态表进行查询
        Table resultTable = table
            .where($("id").isEqual("sensor_1"))
            .select($("id"), $("ts"), $("vc"));
        // 4. 把动态表转换成流
        DataStream<Row> resultStream = tableEnv.toAppendStream(resultTable, Row.class);
        resultStream.print();
        try {
            env.execute();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

2.3 基本使用:聚合操作

```
// 3. 对动态表进行查询
Table resultTable = table
    .where($("vc").isGreaterOrEqual(20))
    .groupBy($("id"))
    .aggregate($("vc").sum().as("vc_sum"))
    .select($("id"), $("vc_sum"));
// 4. 把动态表转换成流 如果涉及到数据的更新, 要用到撤回流. 多个了一个boolean标记
DataStream<Tuple2<Boolean, Row>> resultStream = tableEnv.toRetractStream(resultTable,
Row.class);
```

2.4 表到流的转换

动态表可以像普通数据库表一样通过 INSERT、UPDATE 和 DELETE 来不断修改。它可能是一个只有一行、不断更新的表，也可能是一个 insert-only 的表，没有 UPDATE 和 DELETE 修改，或者介于两者之间的其他表。

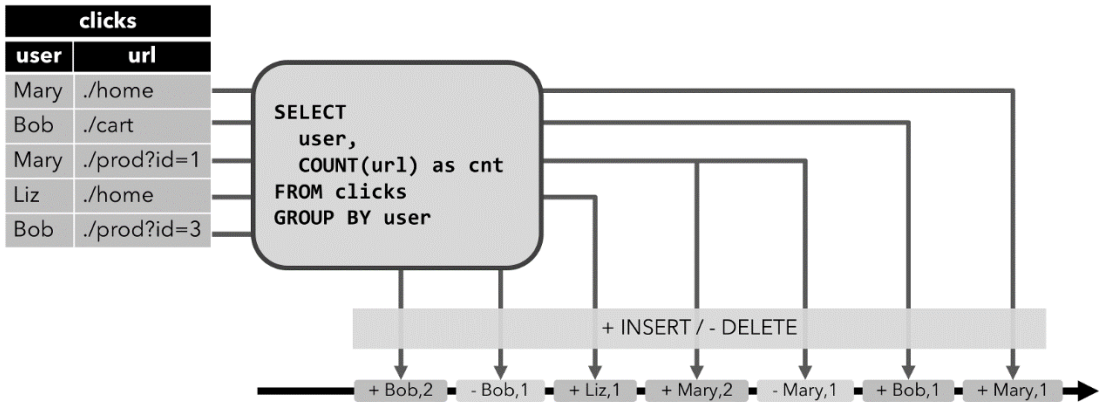
在将动态表转换为流或将其写入外部系统时，需要对这些更改进行编码。Flink 的 Table API 和 SQL 支持三种方式来编码一个动态表的变化:

2.4.1 Append-only 流

仅通过 `INSERT` 操作修改的动态表可以通过输出插入的行转换为流。

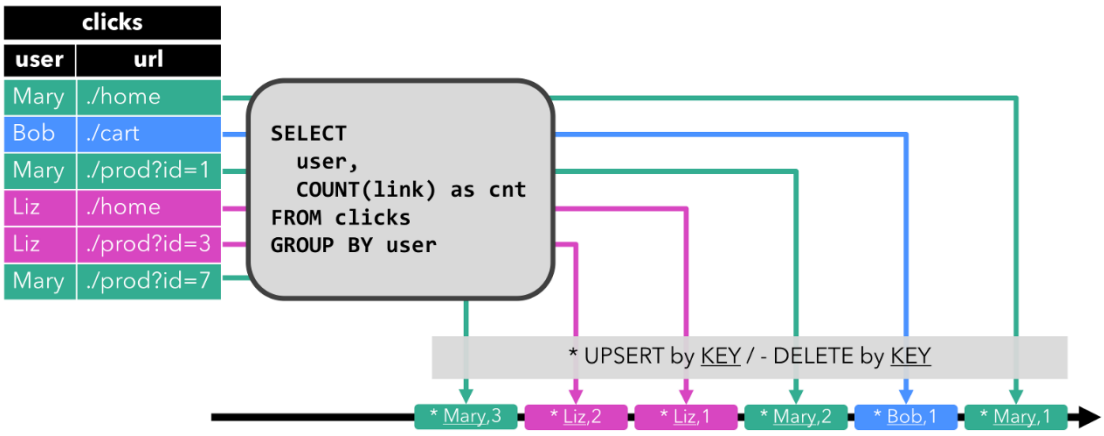
2.4.2 Retract 流

retract 流包含两种类型的 message： *add messages* 和 *retract messages* 。通过将 INSERT 操作编码为 add message、将 DELETE 操作编码为 retract message、将 UPDATE 操作编码为更新(先前)行的 retract message 和更新(新)行的 add message，将动态表转换为 retract 流。下图显示了将动态表转换为 retract 流的过程。



2.4.3 Upsert 流

upsert 流包含两种类型的 message： *upsert messages* 和 *delete messages*。转换为 upsert 流的动态表需要(可能是组合的)唯一键。通过将 INSERT 和 UPDATE 操作编码为 upsert message，将 DELETE 操作编码为 delete message ，将具有唯一键的动态表转换为流。消费流的算子需要知道唯一键的属性，以便正确地应用 message。与 retract 流的主要区别在于 UPDATE 操作是用单个 message 编码的，因此效率更高。下图显示了将动态表转换为 upsert 流的过程。



请注意，在将动态表转换为 DataStream 时，只支持 append 流和 retract 流。

2.5 通过 Connector 声明读入数据

前面是先得到流，再转成动态表，其实动态表也可以直接连接到数据

2.5.1 File source

```
// 2. 创建表
// 2.1 表的元数据信息
Schema schema = new Schema()
    .field("id", DataTypes.STRING())
    .field("ts", DataTypes.BIGINT())
    .field("vc", DataTypes.INT());
// 2.2 连接文件，并创建一个临时表，其实就是一个动态表
tableEnv.connect(new FileSystem().path("input/sensor.txt"))
    .withFormat(new Csv().fieldDelimiter(',').lineDelimiter("\n"))
    .withSchema(schema)
    .createTemporaryTable("sensor");
// 3. 做成表对象，然后对动态表进行查询
Table sensorTable = tableEnv.from("sensor");
Table resultTable = sensorTable
    .groupBy($("id"))
    .select($("id"), $("id").count().as("cnt"));
// 4. 把动态表转换成流. 如果涉及到数据的更新，要用到撤回流. 多个了一个boolean标记
DataStream<Tuple2<Boolean, Row>> resultStream = tableEnv.toRetractStream(resultTable,
Row.class);
resultStream.print();
```

2.5.2 Kafka Source

```
// 2. 创建表
// 2.1 表的元数据信息
Schema schema = new Schema()
    .field("id", DataTypes.STRING())
    .field("ts", DataTypes.BIGINT())
    .field("vc", DataTypes.INT());
// 2.2 连接文件，并创建一个临时表，其实就是一个动态表
tableEnv
    .connect(new Kafka()
                .version("universal")
                .topic("sensor")
                .startFromLatest()
                .property("group.id", "bigdata")
                .property("bootstrap.servers",
"hadoop102:9092,hadoop103:9092,hadoop104:9092"))
    .withFormat(new Json())
    .withSchema(schema)
    .createTemporaryTable("sensor");
// 3. 对动态表进行查询
Table sensorTable = tableEnv.from("sensor");
Table resultTable = sensorTable
    .groupBy($("id"))
    .select($("id"), $("id").count().as("cnt"));
// 4. 把动态表转换成流. 如果涉及到数据的更新，要用到撤回流. 多个了一个boolean标记
DataStream<Tuple2<Boolean, Row>> resultStream = tableEnv.toRetractStream(resultTable,
Row.class);
resultStream.print();
```

2.6 通过 Connector 声明写出数据

2.6.1 File Sink

```
package com.atguigu.flink.java.chapter_11;

import com.atguigu.flink.java.chapter_5.WaterSensor;
import org.apache.flink.streaming.api.datastream.DataStreamSource;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.table.api.DataTypes;
import org.apache.flink.table.api.Table;
import org.apache.flink.table.api.bridge.java.StreamTableEnvironment;
import org.apache.flink.table.descriptors.Csv;
import org.apache.flink.table.descriptors.FileSystem;
import org.apache.flink.table.descriptors.Schema;

import static org.apache.flink.table.api.Expressions.$;

public class Flink02_TableApi_ToFileSystem {
    public static void main(String[] args) {
        StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(1);
        DataStreamSource<WaterSensor> waterSensorStream =
            env.fromElements(new WaterSensor("sensor_1", 1000L, 10),
                             new WaterSensor("sensor_1", 2000L, 20),
                             new WaterSensor("sensor_2", 3000L, 30),
                             new WaterSensor("sensor_1", 4000L, 40),
                             new WaterSensor("sensor_1", 5000L, 50),
                             new WaterSensor("sensor_2", 6000L, 60));
        // 1. 创建表的执行环境
        StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env);

        Table sensorTable = tableEnv.fromDataStream(waterSensorStream);
        Table resultTable = sensorTable
            .where($("id").isEqual("sensor_1") )
            .select($("id"), $("ts"), $("vc"));

        // 创建输出表
        Schema schema = new Schema()
            .field("id", DataTypes.STRING())
            .field("ts", DataTypes.BIGINT())
            .field("vc", DataTypes.INT());
        tableEnv
            .connect(new FileSystem().path("output/sensor_id.txt"))
            .withFormat(new Csv().fieldDelimiter('|'))
            .withSchema(schema)
            .createTemporaryTable("sensor");

        // 把数据写入到输出表中
        resultTable.executeInsert("sensor");
    }
}
```

2.6.2 Kafka Sink

```
package com.atguigu.flink.java.chapter_11;

import com.atguigu.flink.java.chapter_5.WaterSensor;
import org.apache.flink.streaming.api.datastream.DataStreamSource;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.table.api.DataTypes;
import org.apache.flink.table.api.Table;
```

更多 Java –大数据 –前端 –python 人工智能资料下载，可百度访问：尚硅谷官网

```
import org.apache.flink.table.api.bridge.java.StreamTableEnvironment;
import org.apache.flink.table.descriptors.Json;
import org.apache.flink.table.descriptors.Kafka;
import org.apache.flink.table.descriptors.Schema;

import static org.apache.flink.table.api.Expressions.$;

public class Flink03_TableApi_ToKafka {
    public static void main(String[] args) {
        StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(1);
        DataStreamSource<WaterSensor> waterSensorStream =
            env.fromElements(new WaterSensor("sensor_1", 1000L, 10),
                             new WaterSensor("sensor_1", 2000L, 20),
                             new WaterSensor("sensor_2", 3000L, 30),
                             new WaterSensor("sensor_1", 4000L, 40),
                             new WaterSensor("sensor_1", 5000L, 50),
                             new WaterSensor("sensor_2", 6000L, 60));
        // 1. 创建表的执行环境
        StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env);

        Table sensorTable = tableEnv.fromDataStream(waterSensorStream);
        Table resultTable = sensorTable
            .where($("id").isEqual("sensor_1"))
            .select($("id"), $("ts"), $("vc"));

        // 创建输出表
        Schema schema = new Schema()
            .field("id", DataTypes.STRING())
            .field("ts", DataTypes.BIGINT())
            .field("vc", DataTypes.INT());
        tableEnv
            .connect(new Kafka()
                         .version("universal")
                         .topic("sink_sensor")
                         .sinkPartitionerRoundRobin()
                         .property("bootstrap.servers",
"hadoop102:9092,hadoop103:9092,hadoop104:9092"))
            .withFormat(new Json())
            .withSchema(schema)
            .createTemporaryTable("sensor");

        // 把数据写入到输出表中
        resultTable.executeInsert("sensor");
    }
}
```

2.7 其他 Connector 用法

参考官方文档：https://ci.apache.org/projects/flink/flink-docs-release-1.12/zh/dev/table/connect.html

第3章 Flink SQL

3.1 基本使用

3.1.1 查询未注册的表

```
package com.atguigu.flink.java.chapter_11;

import com.atguigu.flink.java.chapter_5.WaterSensor;
import org.apache.flink.streaming.api.datastream.DataStreamSource;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.table.api.Table;
import org.apache.flink.table.api.bridge.java.StreamTableEnvironment;
import org.apache.flink.types.Row;

public class Flink05_SQL_BaseUse {
    public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(1);
        DataStreamSource<WaterSensor> waterSensorStream =
            env.fromElements(new WaterSensor("sensor_1", 1000L, 10),
                             new WaterSensor("sensor_1", 2000L, 20),
                             new WaterSensor("sensor_2", 3000L, 30),
                             new WaterSensor("sensor_1", 4000L, 40),
                             new WaterSensor("sensor_1", 5000L, 50),
                             new WaterSensor("sensor_2", 6000L, 60));

        StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env);
        // 使用sql查询未注册的表
        Table inputTable = tableEnv.fromDataStream(waterSensorStream);
        Table resultTable = tableEnv.sqlQuery("select * from " + inputTable + " where
id='sensor_1'");
        tableEnv.toAppendStream(resultTable, Row.class).print();

        env.execute();
    }
}
```

3.1.2 查询已注册的表

```
package com.atguigu.flink.java.chapter_11;

import com.atguigu.flink.java.chapter_5.WaterSensor;
import org.apache.flink.streaming.api.datastream.DataStreamSource;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.table.api.Table;
import org.apache.flink.table.api.bridge.java.StreamTableEnvironment;
import org.apache.flink.types.Row;

public class Flink05_SQL_BaseUse_2 {
    public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(1);
        DataStreamSource<WaterSensor> waterSensorStream =
            env.fromElements(new WaterSensor("sensor_1", 1000L, 10),
                             new WaterSensor("sensor_1", 2000L, 20),
                             new WaterSensor("sensor_2", 3000L, 30),
                             new WaterSensor("sensor_1", 4000L, 40),
```

```
                        new WaterSensor("sensor_1", 5000L, 50),
                        new WaterSensor("sensor_2", 6000L, 60));

        StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env);
        // 使用sql查询一个已注册的表
        // 1. 从流得到一个表
        Table inputTable = tableEnv.fromDataStream(waterSensorStream);
        // 2. 把注册为一个临时视图
        tableEnv.createTemporaryView("sensor", inputTable);
        // 3. 在临时视图查询数据，并得到一个新表
        Table resultTable = tableEnv.sqlQuery("select * from sensor where id='sensor_1'");
        // 4. 显示resultTable的数据
        tableEnv.toAppendStream(resultTable, Row.class).print();
        env.execute();
    }
}
```

3.2 Kafka 到 Kafka

使用 sql 从 Kafka 读数据，并写入到 Kafka 中

```
package com.atguigu.flink.java.chapter_11;

import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.table.api.bridge.java.StreamTableEnvironment;

public class Flink05_SQL_Kafka2Kafka {
    public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(1);
        StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env);

        // 1. 注册SourceTable: source_sensor
        tableEnv.executeSql("create table source_sensor (id string, ts bigint, vc int) with("
                            + "'connector' = 'kafka',"
                            + "'topic' = 'topic_source_sensor',"
                            + "'properties.bootstrap.servers' =
'hadoop102:9029,hadoop103:9092,hadoop104:9092',"
                            + "'properties.group.id' = 'atguigu',"
                            + "'scan.startup.mode' = 'latest-offset',"
                            + "'format' = 'json'"
                            + ")");

        // 2. 注册SinkTable: sink_sensor
        tableEnv.executeSql("create table sink_sensor(id string, ts bigint, vc int) with("
                            + "'connector' = 'kafka',"
                            + "'topic' = 'topic_sink_sensor',"
                            + "'properties.bootstrap.servers' =
'hadoop102:9029,hadoop103:9092,hadoop104:9092',"
                            + "'format' = 'json'"
                            + ")");

        // 3. 从SourceTable 查询数据, 并写入到 SinkTable
        tableEnv.executeSql("insert into sink_sensor select * from source_sensor where
id='sensor_1'");
    }
}
```

第4章 时间属性

像窗口（在 Table API 和 SQL ）这种基于时间的操作，需要有时间信息。因此，Table API 中的表就需要提供逻辑时间属性来表示时间，以及支持时间相关的操作。

4.1 处理时间

4.1.1 DataStream 到 Table 转换时定义

处理时间属性可以在 schema 定义的时候用.proctime 后缀来定义。时间属性一定不能定义在一个已有字段上，所以它只能定义在 schema 定义的最后

```
StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
env.setParallelism(1);
DataStreamSource<WaterSensor> waterSensorStream =
    env.fromElements(new WaterSensor("sensor_1", 1000L, 10),
                     new WaterSensor("sensor_1", 2000L, 20),
                     new WaterSensor("sensor_2", 3000L, 30),
                     new WaterSensor("sensor_1", 4000L, 40),
                     new WaterSensor("sensor_1", 5000L, 50),
                     new WaterSensor("sensor_2", 6000L, 60));
// 1. 创建表的执行环境
StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env);

// 声明一个额外的字段来作为处理时间字段
Table sensorTable = tableEnv.fromDataStream(waterSensorStream, $("id"), $("ts"), $("vc"),
$("pt").proctime());

sensorTable.print();

env.execute();
```

4.1.2 在创建表的 DDL 中定义

```
package com.atguigu.flink.java.chapter_11;

import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.table.api.TableResult;
import org.apache.flink.table.api.bridge.java.StreamTableEnvironment;

public class Flink06_TableApi_ProcessTime {
    public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(1);
        // 1. 创建表的执行环境
        StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env);
        // 创建表, 声明一个额外的列作为处理时间
        tableEnv.executeSql("create table sensor(id string,ts bigint,vc int,pt_time as
PROCTIME()) with("
                                + "'connector' = 'filesystem',"
                                + "'path' = 'input/sensor.txt',"
                                + "'format' = 'csv'"
                                + ")");

        TableResult result = tableEnv.executeSql("select * from sensor");
```

```
        result.print();
    }
}
```

4.2 事件时间

事件时间允许程序按照数据中包含的时间来处理，这样可以在有乱序或者晚到的数据的情况下产生一致的处理结果。它可以保证从外部存储读取数据后产生可以复现（replayable）的结果。

除此之外，事件时间可以让程序在流式和批式作业中使用同样的语法。在流式程序中的事件时间属性，在批式程序中就是一个正常的时间字段。

为了能够处理乱序的事件，并且区分正常到达和晚到的事件，Flink 需要从事件中获取事件时间并且产生 watermark（watermarks）。

4.2.1 DataStream 到 Table 转换时定义

事件时间属性可以用.rowtime 后缀在定义 DataStream schema 的时候来定义。时间戳和 watermark 在这之前一定是在 DataStream 上已经定义好了。

在从 DataStream 到 Table 转换时定义事件时间属性有两种方式。取决于用 .rowtime 后缀修饰的字段名字是否是已有字段，事件时间字段可以是：

- 在 schema 的结尾追加一个新的字段。
- 替换一个已经存在的字段。

不管在哪种情况下，事件时间字段都表示 DataStream 中定义的事件的时间戳。

```
package com.atguigu.flink.java.chapter_11;

import com.atguigu.flink.java.chapter_5.WaterSensor;
import org.apache.flink.api.common.eventtime.WatermarkStrategy;
import org.apache.flink.streaming.api.datastream.SingleOutputStreamOperator;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.table.api.Table;
import org.apache.flink.table.api.bridge.java.StreamTableEnvironment;

import java.time.Duration;

import static org.apache.flink.table.api.Expressions.$;

public class Flink07_TableApi_EventTime {
    public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(1);
        SingleOutputStreamOperator<WaterSensor> waterSensorStream = env
            .fromElements(new WaterSensor("sensor_1", 1000L, 10),
                          new WaterSensor("sensor_1", 2000L, 20),
                          new WaterSensor("sensor_2", 3000L, 30),
                          new WaterSensor("sensor_1", 4000L, 40),
                          new WaterSensor("sensor_1", 5000L, 50),
                          new WaterSensor("sensor_2", 6000L, 60))
            .assignTimestampsAndWatermarks(
                WatermarkStrategy
```

```
                .<WaterSensor>forBoundedOutOfOrderness(Duration.ofSeconds(5))
                  .withTimestampAssigner((element, recordTimestamp) -> element.getTs())
            );

        StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env);
        Table table = tableEnv
            // 用一个额外的字段作为事件时间属性
            .fromDataStream(waterSensorStream, $("id"), $("ts"), $("vc"), $("et").rowtime());
        table.execute().print();
        env.execute();

    }
}
```

```
// 使用已有的字段作为时间属性
.fromDataStream(waterSensorStream, $("id"), $("ts").rowtime(), $("vc"));
```

4.2.2 在创建表的 DDL 中定义

事件时间属性可以用 WATERMARK 语句在 CREATE TABLE DDL 中进行定义。WATERMARK 语句在一个已有字段上定义一个 watermark 生成表达式，同时标记这个已有字段为时间属性字段。

```
package com.atguigu.flink.java.chapter_11;

import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.table.api.bridge.java.StreamTableEnvironment;

public class Flink07_TableApi_EventTime_2 {
    public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(1);

        StreamTableEnvironment tEnv = StreamTableEnvironment.create(env);
        // 作为事件时间的字段必须是 timestamp 类型, 所以根据 long 类型的 ts 计算出来一个 t
        tEnv.executeSql("create table sensor(" +
                        "id string," +
                        "ts bigint," +
                        "vc int, " +
                        "t as to_timestamp(from_unixtime(ts/1000,'yyyy-MM-dd HH:mm:ss'))," +
                        "watermark for t as t - interval '5' second)" +
                        "with("
                          + "'connector' = 'filesystem',"
                          + "'path' = 'input/sensor.txt',"
                          + "'format' = 'csv'"
                          + ")");

        tEnv.sqlQuery("select * from sensor").execute().print();

    }
}
```

说明：

1. 把一个现有的列定义为一个为表标记事件时间的属性。该列的类型必须为 **TIMESTAMP(3)**，且是 schema 中的顶层列，它也可以是一个计算列。
2. 严格递增时间戳： `WATERMARK FOR rowtime_column AS rowtime_column`。
3. 递增时间戳：WATERMARK FOR rowtime_column AS rowtime_column - INTERVAL '0.001'

更多 Java –大数据 –前端 –python 人工智能资料下载，可百度访问：尚硅谷官网

SECOND。

4. 有界乱序时间戳： WATERMARK FOR rowtime_column AS rowtime_column - INTERVAL 'string' timeUnit。

第5章 窗口(window)

时间语义，要配合窗口操作才能发挥作用。最主要的用途，当然就是开窗口然后根据时间段做计算了。

下面我们就来看看 Table API 和 SQL 中，怎么利用时间字段做窗口操作。

在 Table API 和 SQL 中，主要有两种窗口：Group Windows 和 Over Windows。

5.1 Table API 中使用窗口

5.1.1 Group Windows

分组窗口（Group Windows）会根据时间或行计数间隔，将行聚合到有限的组（Group）中，并对每个组的数据执行一次聚合函数。

Table API 中的 Group Windows 都是使用。Window（w:GroupWindow）子句定义的，并且必须由 as 子句指定一个别名。为了按窗口对表进行分组，窗口的别名必须在 group by 子句中，像常规的分组字段一样引用。

➢ 滚动窗口

```
public class Flink08_TableApi_Window_1 {
    public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(1);
        SingleOutputStreamOperator<WaterSensor> waterSensorStream = env
            .fromElements(new WaterSensor("sensor_1", 1000L, 10),
                          new WaterSensor("sensor_1", 2000L, 20),
                          new WaterSensor("sensor_2", 3000L, 30),
                          new WaterSensor("sensor_1", 4000L, 40),
                          new WaterSensor("sensor_1", 5000L, 50),
                          new WaterSensor("sensor_2", 6000L, 60))
            .assignTimestampsAndWatermarks(
                WatermarkStrategy
                  .<WaterSensor>forBoundedOutOfOrderness(Duration.ofSeconds(5))
                  .withTimestampAssigner((element, recordTimestamp) -> element.getTs())
            );

        StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env);
        Table table = tableEnv
            .fromDataStream(waterSensorStream, $("id"), $("ts").rowtime(), $("vc"));

        table
```

```
            .window(Tumble.over(lit(10).second()).on($("ts")).as("w"))  // 定义滚动窗口并给窗口起一个别名
            .groupBy($("id"), $("w")) // 窗口必须出现的分组字段中
             .select($("id"), $("w").start(), $("w").end(), $("vc").sum())
             .execute()
             .print();

        env.execute();

    }
}
```

- 滑动窗口

```
.window(Slide.over(lit(10).second()).every(lit(5).second()).on($("ts")).as("w"))
```

- 会话窗口

```
.window(Session.withGap(lit(6).second()).on($("ts")).as("w")
```

5.1.2 Over Windows

Over window 聚合是标准 SQL 中已有的（Over 子句），可以在查询的 SELECT 子句中定义。Over window 聚合，会针对每个输入行，计算相邻行范围内的聚合。

Table API 提供了 Over 类，来配置 Over 窗口的属性。可以在事件时间或处理时间，以及指定为时间间隔、或行计数的范围内，定义 Over windows。

无界的 over window 是使用常量指定的。也就是说，时间间隔要指定 UNBOUNDED_RANGE，或者行计数间隔要指定 UNBOUNDED_ROW。而有界的 over window 是用间隔的大小指定的。

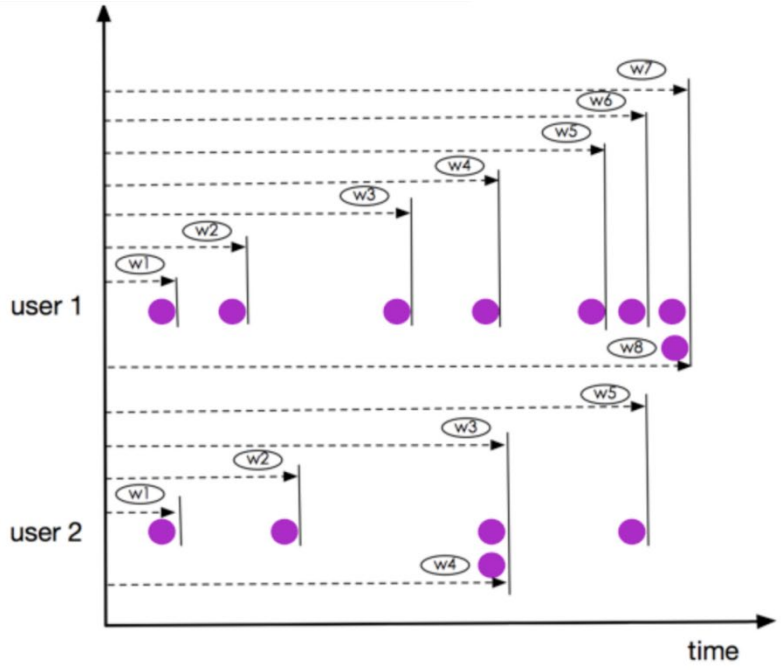
- Unbounded Over Windows

```
public class Flink09_TableApi_OverWindow_1 {
    public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(1);
        SingleOutputStreamOperator<WaterSensor> waterSensorStream = env
            .fromElements(new WaterSensor("sensor_1", 1000L, 10),
                          new WaterSensor("sensor_1", 4000L, 40),
                          new WaterSensor("sensor_1", 2000L, 20),
                          new WaterSensor("sensor_2", 3000L, 30),
                          new WaterSensor("sensor_1", 5000L, 50),
                          new WaterSensor("sensor_2", 6000L, 60))
             .assignTimestampsAndWatermarks(
                 WatermarkStrategy
                   .<WaterSensor>forBoundedOutOfOrderness(Duration.ofSeconds(1))
                    .withTimestampAssigner((element, recordTimestamp) -> element.getTs())
             );

        StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env);
```

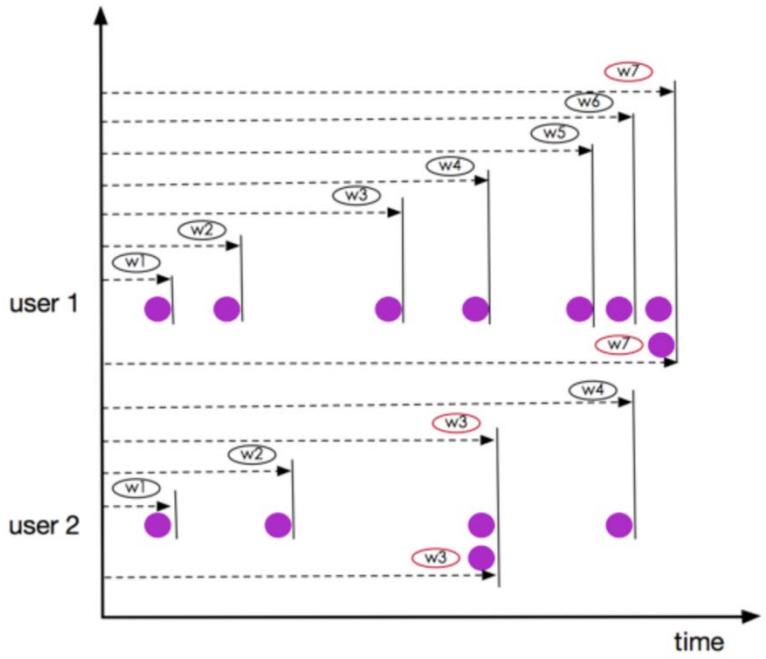
```
        Table table = tableEnv
            .fromDataStream(waterSensorStream, $("id"), $("ts").rowtime(), $("vc"));

        table

.window(Over.partitionBy($("id")).orderBy($("ts")).preceding(UNBOUNDED_ROW).as("w"))
            .select($("id"), $("ts"), $("vc").sum().over($("w")).as("sum_vc"))
            .execute()
            .print();

        env.execute();

    }

}

# 使用UNBOUNDED_RANGE
.window(Over.partitionBy($("id")).orderBy($("ts")).preceding(UNBOUNDED_RANGE).as("w"))
```

说明：

正交属性	说明	proctime	eventtime
ROWS OVER Window	按照实际元素的行确定窗口。	支持	支持
RANGE OVER Window	按照实际的元素值（时间戳值）确定窗口。	支持	支持



说明 虽然上图所示窗口user1的w7、w8及user2的窗口w3、w4都是同一时刻到达，但它们仍然在不同的窗口，这一点与RANGE OVER Window不同。



说明 上图所示窗口user1的w7、user2的窗口w3，两个元素同一时刻到达，属于相同的window，这一点与ROWS OVER Window不同。

➢ Bounded Over Windows

```
// 当事件时间向前算3s得到一个窗口
.window(Over.partitionBy($("id")).orderBy($("ts")).preceding(lit(3).second()).as("w"))
// 当行向前推算2行算一个窗口
.window(Over.partitionBy($("id")).orderBy($("ts")).preceding(rowInterval(2L)).as("w"))
```

5.2 SQL API 中使用窗口

5.2.1 Group Windows

SQL 查询的分组窗口是通过 GROUP BY 子句定义的。类似于使用常规 GROUP BY 语句的查询，窗口分组语句的 GROUP BY 子句中带有一个窗口函数为每个分组计算出一个结果。以下是批处理表和流处理表支持的分组窗口函数：

Group Window Function	Description
`TUMBLE(time_attr, interval)`	Defines a tumbling time window. A tumbling time window assigns rows to non-overlapping, continuous windows with a fixed duration (`interval`). For example, a tumbling window of 5 minutes groups rows in 5 minutes intervals. Tumbling windows can be defined on event-time (stream + batch) or processing-time (stream).
`HOP(time_attr, interval, interval)`	Defines a hopping time window (called sliding window in the Table API). A hopping time window has a fixed duration (second `interval` parameter) and hops by a specified hop interval (first `interval` parameter). If the hop interval is smaller than the window size, hopping windows are overlapping. Thus, rows can be assigned to multiple windows. For example, a hopping window of 15 minutes size and 5 minute hop interval assigns each row to 3 different windows of 15 minute size, which are evaluated in an interval of 5 minutes. Hopping windows can be defined on event-time (stream + batch) or processing-time (stream).
`SESSION(time_attr, interval)`	Defines a session time window. Session time windows do not have a fixed duration but their bounds are defined by a time `interval` of inactivity, i.e., a session window is closed if no event appears for a defined gap period. For example a session window with a 30 minute gap starts when a row is observed after 30 minutes inactivity (otherwise the row would be added to an existing window) and is closed if no row is added within 30 minutes. Session windows can work on event-time (stream + batch) or processing-time (stream).

可以使用以下辅助函数选择组窗口的开始和结束时间戳以及时间属性：

辅助函数	描述

`TUMBLE_START(time_attr, interval)`
`HOP_START(time_attr, interval, interval)`
`SESSION_START(time_attr, interval)`
	返回相对应的滚动、滑动和会话窗口范围内的下界时间戳。

`TUMBLE_END(time_attr, interval)`
`HOP_END(time_attr, interval, interval)`
`SESSION_END(time_attr, interval)`
	返回相对应的滚动、滑动和会话窗口*范围以外*的上界时间戳。
	注意： 范围以外的上界时间戳*不可以* 在随后基于时间的操作中，作为 行时间属性 使用，比如 interval join 以及 分组窗口或分组窗口上的聚合。

`TUMBLE_ROWTIME(time_attr, interval)`
`HOP_ROWTIME(time_attr, interval, interval)`
`SESSION_ROWTIME(time_attr, interval)`
	返回相对应的滚动、滑动和会话窗口*范围以内*的上界时间戳。
	返回的是一个可用于后续需要基于时间的操作的时间属性（rowtime attribute），比如interval join 以及 分组窗口或分组窗口上的聚合。

`TUMBLE_PROCTIME(time_attr, interval)`
`HOP_PROCTIME(time_attr, interval, interval)`
`SESSION_PROCTIME(time_attr, interval)`
	返回一个可用于后续需要基于时间的操作的 处理时间参数，比如interval join 以及 分组窗口或分组窗口上的聚合.

```
StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
env.setParallelism(1);
StreamTableEnvironment tEnv = StreamTableEnvironment.create(env);

// 作为事件时间的字段必须是 timestamp 类型, 所以根据 long 类型的 ts 计算出来一个 t
tEnv.executeSql("create table sensor(" +
                "id string," +
                "ts bigint," +
                "vc int, " +
                "t as to_timestamp(from_unixtime(ts/1000,'yyyy-MM-dd HH:mm:ss'))," +
                "watermark for t as t - interval '5' second)" +
                "with("
                  + "'connector' = 'filesystem',"
                  + "'path' = 'input/sensor.txt',"
                  + "'format' = 'csv'"
                  + ")");

tEnv
    .sqlQuery(
        "SELECT id, " +
          "  TUMBLE_START(t, INTERVAL '1' minute) as wStart,  " +
          "  TUMBLE_END(t, INTERVAL '1' minute) as wEnd,  " +
```

```
            "  SUM(vc) sum_vc " +
            "FROM sensor " +
            "GROUP BY TUMBLE(t, INTERVAL '1' minute), id"
    )
    .execute()
    .print();
```

```
tEnv
    .sqlQuery(
        "SELECT id, " +
            "  hop_start(t, INTERVAL '1' minute, INTERVAL '1' hour) as wStart,  " +
            "  hop_end(t, INTERVAL '1' minute, INTERVAL '1' hour) as wEnd,  " +
            "  SUM(vc) sum_vc " +
            "FROM sensor " +
            "GROUP BY hop(t, INTERVAL '1' minute, INTERVAL '1' hour), id"
    )
    .execute()
    .print();
```

5.2.2 Over Windows

```
tEnv
    .sqlQuery(
        "select " +
            "id," +
            "vc," +
            "sum(vc) over(partition by id order by t rows between 1 PRECEDING and current row)"
            + "from sensor"
    )
    .execute()
    .print();
```

```
tEnv
    .sqlQuery(
        "select " +
            "id," +
            "vc," +
            "count(vc) over w, " +
            "sum(vc) over w " +
            "from sensor " +
            "window w as (partition by id order by t rows between 1 PRECEDING and current row)"
    )
    .execute()
    .print();
```

第6章 函数（Functions）

Flink Table 和 SQL 内置了很多 SQL 中支持的函数；如果有无法满足的需要，则可以实现用户自定义的函数（UDF）来解决。

6.1 系统内置函数

Flink Table API 和 SQL 为用户提供了一组用于数据转换的内置函数。SQL 中支持的

很多函数，Table API 和 SQL 都已经做了实现，其它还在快速开发扩展中。

以下是一些典型函数的举例，全部的内置函数，可以参考官网介绍。

- 比较函数

SQL：

```
value1 = value2
```

```
value1 > value2
```

Table API：

```
ANY1 === ANY2
```

```
ANY1 > ANY2
```

- 逻辑函数

SQL：

```
boolean1 OR boolean2
```

```
boolean IS FALSE
```

```
NOT boolean
```

Table API：

```
BOOLEAN1 || BOOLEAN2
```

```
BOOLEAN.isFalse
```

```
!BOOLEAN
```

- 算术函数

SQL：

```
numeric1 + numeric2
```

```
POWER(numeric1, numeric2)
```

Table API：

```
NUMERIC1 + NUMERIC2
```

```
NUMERIC1.power(NUMERIC2)
```

- 字符串函数

SQL：

```
string1 || string2
```

```
UPPER(string)
```

```
CHAR_LENGTH(string)
```

Table API：

```
STRING1 + STRING2
```

```
STRING.upperCase()
```

```
STRING.charLength()
```

- 时间函数

SQL：

```
DATE string
```

```
TIMESTAMP string
```

```
CURRENT_TIME
```

```
INTERVAL string range
```

Table API：

```
STRING.toDate
```

```
STRING.toTimestamp
```

更多 Java –大数据 –前端 –python 人工智能资料下载，可百度访问：尚硅谷官网

`currentTime()`

`NUMERIC.days`

`NUMERIC.minutes`

- 聚合函数

SQL：

`COUNT(*)`

`SUM([ ALL | DISTINCT ] expression)`

`RANK()`

`ROW_NUMBER()`

Table API：

`FIELD.count`

`FIELD.sum0`

6.2 UDF

用户定义函数（User-defined Functions，UDF）是一个重要的特性，因为它们显著地扩展了查询（Query）的表达能力。一些系统内置函数无法解决的需求，我们可以用 UDF 来自定义实现。

6.2.1 注册用户自定义函数 UDF

在大多数情况下，用户定义的函数必须先注册，然后才能在查询中使用。不需要专门为 Scala 的 Table API 注册函数。

函数通过调用 registerFunction()方法在 TableEnvironment 中注册。当用户定义的函数被注册时，它被插入到 TableEnvironment 的函数目录中，这样 Table API 或 SQL 解析器就可以识别并正确地解释它。

6.2.2 标量函数（Scalar Functions）

用户定义的标量函数，可以将 0、1 或多个标量值，映射到新的标量值。

为了定义标量函数，必须在 org.apache.flink.table.functions 中扩展基类 Scalar Function，并实现（一个或多个）求值（evaluation，eval）方法。标量函数的行为由求值方法决定，求值方法必须公开声明并命名为 eval（直接 def 声明，没有 override）。求值方法的参数类型和返回类型，确定了标量函数的参数和返回类型。

在下面的代码中，我们定义自己的 HashCode 函数，在 TableEnvironment 中注册它，并在查询中调用它。

```
// 自定义一个标量函数
public static class HashCode extends ScalarFunction {
    private int factor = 13;

    public HashCode(int factor) {
        this.factor = factor;
    }

    public int eval(String s) {
        return s.hashCode() * factor;
    }
}
```

主函数中调用，计算 sensor id 的哈希值（前面部分照抄，流环境、表环境、读取 source、建表）：

```
public static void main(String[] args) throws Exception {
    // 1. 创建环境
    StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
    env.setParallelism(1);

    StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env);

    // 2. 读取文件，得到 DataStream
    String filePath = "sensor";

    DataStream<String> inputStream = env.readTextFile(filePath);

    // 3. 转换成 Java Bean，并指定timestamp 和watermark
    DataStream<SensorReading> dataStream = inputStream
            .map( line -> {
                String[] fields = line.split(",");
                return new SensorReading(fields[0], new Long(fields[1]), new
Double(fields[2]));
            } );

    // 4. 将 DataStream 转换为 Table
    Table sensorTable = tableEnv.fromDataStream(dataStream, "id, timestamp
as ts, temperature");

    // 5. 调用自定义 hash 函数，对 id 进行 hash 运算
    HashCode hashCode = new HashCode(23);
    tableEnv.registerFunction("hashCode", hashCode);
```

```
    Table resultTable = sensorTable
            .select("id, ts, hashCode(id)");

    // sql
    tableEnv.createTemporaryView("sensor", sensorTable);
    Table resultSqlTable = tableEnv.sqlQuery("select id, ts, hashCode(id)
from sensor");

    tableEnv.toAppendStream(resultTable, Row.class).print("result");
    tableEnv.toRetractStream(resultSqlTable, Row.class).print("sql");

    env.execute("scalar function test");
}
```

6.2.3 表函数（Table Functions）

与用户定义的标量函数类似，用户定义的表函数，可以将 0、1 或多个标量值作为输入参数；与标量函数不同的是，它可以返回任意数量的行作为输出，而不是单个值。

为了定义一个表函数，必须扩展 org.apache.flink.table.functions 中的基类 TableFunction 并实现（一个或多个）求值方法。表函数的行为由其求值方法决定，求值方法必须是 public 的，并命名为 eval。求值方法的参数类型，决定表函数的所有有效参数。

返回表的类型由 TableFunction 的泛型类型确定。求值方法使用 protected collect（T）方法发出输出行。

在 Table API 中，Table 函数需要与.joinLateral 或.leftOuterJoinLateral 一起使用。

joinLateral 算子，会将外部表中的每一行，与表函数（TableFunction，算子的参数是它的表达式）计算得到的所有行连接起来。

而 leftOuterJoinLateral 算子，则是左外连接，它同样会将外部表中的每一行与表函

更多 Java –大数据 –前端 –python 人工智能资料下载，可百度访问：尚硅谷官网

数计算生成的所有行连接起来；并且，对于表函数返回的是空表的外部行，也要保留下来。

在 SQL 中，则需要使用 Lateral Table（<TableFunction>），或者带有 ON TRUE 条件的左连接。

下面的代码中，我们将定义一个表函数，在表环境中注册它，并在查询中调用它。

自定义 TableFunction：

```
// 自定义 TableFunction
public static class Split extends TableFunction<Tuple2<String, Integer>> {
    private String separator = ",";

    public Split(String separator) {
        this.separator = separator;
    }

    // 类似 flatmap，没有返回值
    public void eval(String str) {
        for (String s : str.split(separator)) {
            collect(new Tuple2<String, Integer>(s, s.length()));
        }
    }
}
```

接下来，就是在代码中调用。首先是 Table API 的方式：

```
Split split = new Split("_");
tableEnv.registerFunction("split", split);
Table resultTable = sensorTable
```

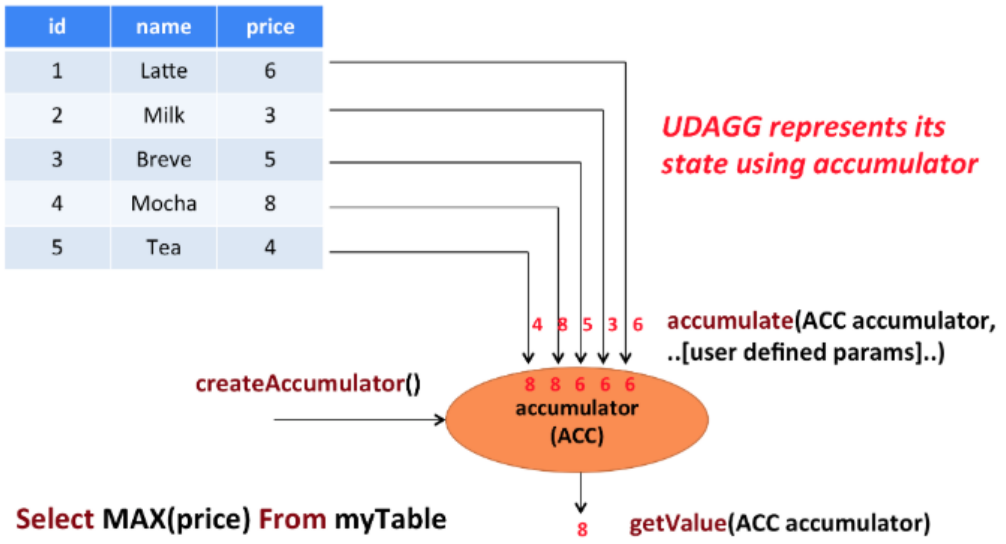
```
        .joinLateral( "split(id) as (word, length)")
        .select("id, ts, word, length");
```

然后是 SQL 的方式：

```
tableEnv.createTemporaryView("sensor", sensorTable);
Table resultSqlTable = tableEnv.sqlQuery("select id, ts, word, length " +
        "from sensor, lateral table( split(id) ) as splitId(word, length)");
```

6.2.4 聚合函数（Aggregate Functions）

用户自定义聚合函数（User-Defined Aggregate Functions，UDAGGs）可以把一个表中的数据，聚合成一个标量值。用户定义的聚合函数，是通过继承 AggregateFunction 抽象类实现的。



上图中显示了一个聚合的例子。

假设现在有一张表，包含了各种饮料的数据。该表由三列（id、name 和 price）、五行组成数据。现在我们需要找到表中所有饮料的最高价格，即执行 max（）聚合，结果将是一

个数值。

AggregateFunction 的工作原理如下。

- 首先，它需要一个累加器，用来保存聚合中间结果的数据结构（状态）。可以通过调用 AggregateFunction 的 createAccumulator（）方法创建空累加器。
- 随后，对每个输入行调用函数的 accumulate（）方法来更新累加器。
- 处理完所有行后，将调用函数的 getValue（）方法来计算并返回最终结果。

AggregationFunction 要求必须实现的方法：

- createAccumulator()
- accumulate()
- getValue()

除了上述方法之外，还有一些可选择实现的方法。其中一些方法，可以让系统执行查询更有效率，而另一些方法，对于某些场景是必需的。例如，如果聚合函数应用在会话窗口（session group window）的上下文中，则 merge（）方法是必需的。

- retract()
- merge()
- resetAccumulator()

接下来我们写一个自定义 AggregateFunction，计算一下每个 sensor 的平均温度值。

```
// 定义 AggregateFunction 的 Accumulator
public static class AvgTempAcc {
    double sum = 0.0;
    int count = 0;
}

// 自定义一个聚合函数，求每个传感器的平均温度值，保存状态(tempSum, tempCount)
```

```
public static class AvgTemp extends AggregateFunction<Double, AvgTempAcc>{

    @Override

    public Double getValue(AvgTempAcc accumulator) {

        return accumulator.sum / accumulator.count;

    }


    @Override

    public AvgTempAcc createAccumulator() {

        return new AvgTempAcc();

    }


    // 实现一个具体的处理计算函数，accumulate

    public void accumulate( AvgTempAcc accumulator, Double temp) {

        accumulator.sum += temp;

        accumulator.count += 1;

    }

}
```

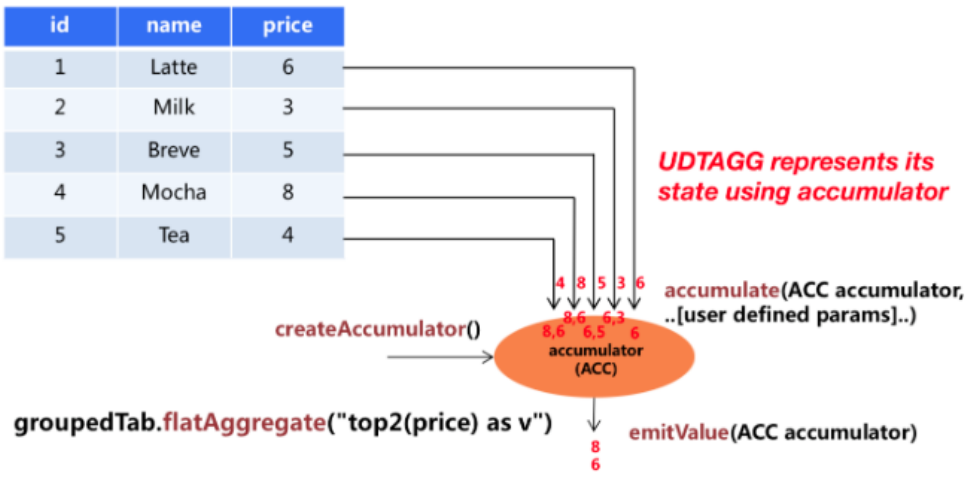
接下来就可以在代码中调用了。

```
// 创建一个聚合函数实例

AvgTemp avgTemp = new AvgTemp();



// Table API 的调用

tableEnv.registerFunction("avgTemp", avgTemp);

Table resultTable = sensorTable

        .groupBy("id")

        .aggregate("avgTemp(temperature) as avgTemp")

        .select("id, avgTemp");
```

```
// sql
tableEnv.createTemporaryView("sensor", sensorTable);
Table resultSqlTable = tableEnv.sqlQuery("select id, avgTemp(temperature) "
+
        "from sensor group by id");

tableEnv.toRetractStream(resultTable, Row.class).print("result");
tableEnv.toRetractStream(resultSqlTable, Row.class).print("sql");
```

6.2.5 表聚合函数（Table Aggregate Functions）

用户定义的表聚合函数（User-Defined Table Aggregate Functions，UDTAGGs），可以把一个表中数据，聚合为具有多行和多列的结果表。这跟 AggregateFunction 非常类似，只是之前聚合结果是一个标量值，现在变成了一张表。



比如现在我们需要找到表中所有饮料的前 2 个最高价格，即执行 top2（）表聚合。我们需要检查 5 行中的每一行，得到的结果将是一个具有排序后前 2 个值的表。

用户定义的表聚合函数，是通过继承 TableAggregateFunction 抽象类来实现的。

TableAggregateFunction 的工作原理如下。

- 首先，它同样需要一个累加器(Accumulator)，它是保存聚合中间结果的数据结构。通过调用 TableAggregateFunction 的 createAccumulator（）方法可以创建空累加器。
- 随后，对每个输入行调用函数的 accumulate（）方法来更新累加器。
- 处理完所有行后，将调用函数的 emitValue（）方法来计算并返回最终结果。

AggregationFunction 要求必须实现的方法：

- createAccumulator()
- accumulate()

除了上述方法之外，还有一些可选择实现的方法。

- retract()
- merge()
- resetAccumulator()
- emitValue()
- emitUpdateWithRetract()

接下来我们写一个自定义 TableAggregateFunction，用来提取每个 sensor 最高的两个温度值。

```
// 先定义一个 Accumulator
public static class Top2TempAcc {
    double highestTemp = Double.MIN_VALUE;
    double secondHighestTemp = Double.MIN_VALUE;
}
```

```
// 自定义表聚合函数
public static class Top2Temp extends TableAggregateFunction<Tuple2<Double,
Integer>, Top2TempAcc> {
    @Override
    public Top2TempAcc createAccumulator() {
        return new Top2TempAcc();
    }

    // 实现计算聚合结果的函数 accumulate
    public void accumulate(Top2TempAcc acc, Double temp) {
        if (temp > acc.highestTemp) {
            acc.secondHighestTemp = acc.highestTemp;
            acc.highestTemp = temp;
        } else if (temp > acc.secondHighestTemp) {
            acc.secondHighestTemp = temp;
        }
    }
    // 实现一个输出结果的方法，最终处理完表中所有数据时调用
    public void emitValue(Top2TempAcc acc, Collector<Tuple2<Double,
Integer>> out) {
        out.collect(new Tuple2<>(acc.highestTemp, 1));
        out.collect(new Tuple2<>(acc.secondHighestTemp, 2));
    }
}
```

接下来就可以在代码中调用了。

```
// 创建一个表聚合函数实例
Top2Temp top2Temp = new Top2Temp();
tableEnv.registerFunction("top2Temp", top2Temp);
```

```
Table resultTable = sensorTable
        .groupBy("id")
        .flatAggregate("top2Temp(temperature) as (temp, rank)")
        .select("id, temp, rank");


tableEnv.toRetractStream(resultTable, Row.class).print("result");
```

第7章 Catalog

Catalog 提供了**元数据信息**，例如数据库、表、分区、视图以及数据库或其他外部系统中存储的函数和信息。

数据处理最关键的方面之一是**管理元数据**。元数据可以是临时的，例如临时表、或者通过 TableEnvironment 注册的 UDF。元数据也可以是持久化的，例如 Hive Metastore 中的元数据。Catalog 提供了一个统一的 API，用于管理元数据，并使其可以从 Table API 和 SQL 查询语句中来访问。

前面用到 Connector 其实就是在使用 Catalog。

7.1 Catalog 类型

7.1.1 GenericInMemoryCatalog

GenericInMemoryCatalog 是基于内存实现的 Catalog，所有元数据只在 session 的生命周期内可用。

7.1.2 JdbcCatalog

JdbcCatalog 使得用户可以将 Flink 通过 JDBC 协议连接到关系数据库。PostgresCatalog 是当前实现的唯一一种 JDBC Catalog。

7.1.3 HiveCatalog

HiveCatalog 有两个用途：作为原生 Flink 元数据的持久化存储，以及作为读写现有 Hive 元数据的接口。 Flink 的 Hive 文档提供了有关设置 HiveCatalog 以及访问现有 Hive 元数据的详细信息。

更多 Java –大数据 –前端 –python 人工智能资料下载，可百度访问：尚硅谷官网

7.2 HiveCatalog

7.2.1 导入需要的依赖

```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-connector-hive_${scala.binary.version}</artifactId>
    <version>${flink.version}</version>
</dependency>
<!-- Hive Dependency -->
<dependency>
    <groupId>org.apache.hive</groupId>
    <artifactId>hive-exec</artifactId>
    <version>3.1.2</version>
</dependency>
```

7.2.2 在 hadoop102 启动 hive 元数据

```
nohup hive --service metastore >/dev/null 2>&1 &
```

7.2.3 连接 Hive

```
StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
env.setParallelism(1);
StreamTableEnvironment tEnv = StreamTableEnvironment.create(env);

String name            = "myhive";  // Catalog 名字
String defaultDatabase = "flink_test"; // 默认数据库
String hiveConfDir     = "c:/conf"; // hive配置文件的目录. 需要把hive-site.xml添加到该目录


// 1. 创建HiveCatalog
HiveCatalog hive = new HiveCatalog(name, defaultDatabase, hiveConfDir);
// 2. 注册HiveCatalog
tEnv.registerCatalog(name, hive);
// 3. 把 HiveCatalog: myhive 作为当前session的catalog
tEnv.useCatalog(name);
tEnv.useDatabase("flink_test");
tEnv.sqlQuery("select * from stu").execute().print();
```

第8章 SqlClient

启动换一个 yarn-session，然后启动换一个 sql 客户端。

```
bin/sql-client.sh embedded
```

8.1 建立到 Kafka 的连接

下面创建一个流表从 Kafka 读取数据

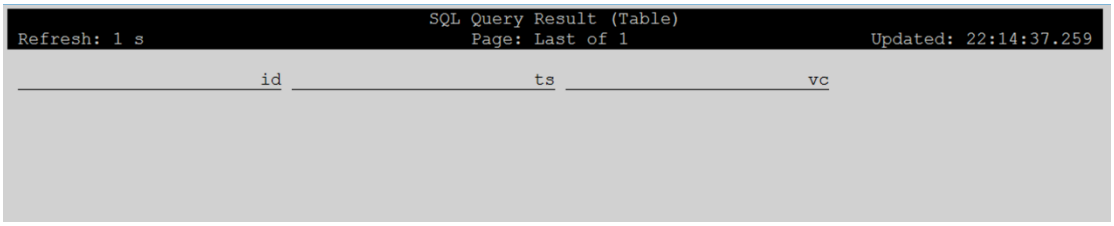
copy 依赖到 flink 的 lib 目录下 flink-sql-connector-kafka_2.11-1.12.0.jar

下载地址：https://repo.maven.apache.org/maven2/org/apache/flink/flink-sql-connector-kafka_2.11/1.12.0/flink-sql-connector-kafka_2.11-1.12.0.jar

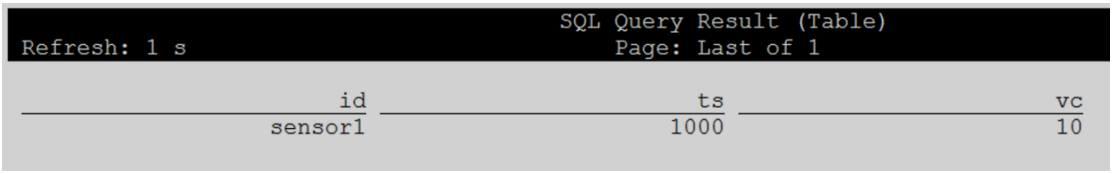
```
create table sensor(id string, ts bigint, vc int)
with(
    'connector'='kafka',
    'topic'='flink_sensor',
    'properties.bootstrap.servers'='hadoop102:9092',
    'properties.group.id'='atguigu',
    'format'='json',
    'scan.startup.mode'='latest-offset'
)
```

从流标查询数据

```
select * from sensor;
```



向 Kafka 写入数据：`{"id": "sensor1", "ts": 1000, "vc": 10}`



8.2 建立到 mysql 的连接

依赖：flink-sql-connector-kafka_2.11-1.12.0.ja

下载地址：https://repo.maven.apache.org/maven2/org/apache/flink/flink-sql-connector-kafka_2.11/1.12.0/flink-sql-connector-kafka_2.11-1.12.0.ja

copy mysql 驱动到 lib 目录

```
create table sensor(id string, ts bigint, vc int)
with(
    'connector' = 'jdbc',
    'url' = 'jdbc:mysql://hadoop102:3306/gmall',
    'username'='root',
    'password'='aaaaaa',
    'table-name' = 'sensor'
)
```